The Report Committee for Abigail Lauren Dowd
Certifies that this is the approved version of the following Report:

Performance of Java in Function-as-a-Service Computing

APPROVED BY
SUPERVISING COMMITTEE:

Lizy Kurian John, Supervisor

Christine Julien

Performance of Java in Function-as-a-Service Computing

by

Abigail Lauren Dowd

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2020

Acknowledgements

I would like to thank my advisor, Dr. Lizy John, for her guidance and support throughout this year. She helped to give my project direction when it was needed, while allowing me the freedom to pursue what interested me most. I also thank the members of the Laboratory for Computer Architecture (LCA), who provided valuable feedback and technical insight throughout my project.

I would like to thank my friends and family for their much-needed love and support throughout these two years at the University of Texas at Austin.

Abstract

Performance of Java in Function-as-a-Service Computing

Abigail Lauren Dowd, MSE

The University of Texas at Austin, 2020

Supervisor: Lizy Kurian John

Over the past several years serverless computing has rapidly grown in popularity because of its flexibility and low cost. Despite the name, serverless computing is not actually serverless. It simply means developers do not need to be aware of the servers because resources are allocated as needed, and developers are only charged for what they use. One of the newest forms of serverless computing is Function-as-a-Service (FaaS). FaaS provides a framework for executing modular pieces of code in response to an event, such as a user clicking a button in a web application. Developers using FaaS are only charged for the execution time of their functions. Although FaaS has many apparent benefits, relatively little is known about the performance of FaaS. Past work has shown that cold starts typically have a negative effect on latency, but the magnitude of the slowdown varies depending on provider and language [13, 16]. Sharad et al. recently performed a study on the Apache OpenWhisk FaaS platform and found that for functions written in Python, there is a significant slowdown compared to native execution and that the cold start time can be up to 10x the execution time of a short function [24]. The purpose of this report is to investigate the overhead due to containerization and cold starts for

functions written in Java to determine whether the findings stated above hold regardless of language. I found that the container initialization time for Java was consistently less than half that of Python. However, Java has an additional overhead due to Java Virtual Machine (JVM) warmup which contributes to the execution time in a cold start scenario. Java functions which caused a cold start showed up to a 50x slowdown compared to native execution for very short functions and a 19x slowdown on average. Java functions which executed in a warm environment still had a significant overhead with a 5x slowdown on average. Overall, Java functions had a greater containerization overhead than Python functions. However, Java still had a faster execution time than Python on average—3.6x and 6.7x faster for warm and cold starts, respectively.

Table of Contents

List of Tables

List of Figures

Chapter I: Introduction

OVERVIEW

Over the past several years, interest in serverless computing has rapidly increased due to its flexibility and low cost. Despite its name, serverless computing is not actually serverless. It simply adds a layer of abstraction between the server and the developer. So, what exactly is serverless? Eyk et al. give the following definition: "Serverless Computing is a form of cloud computing which allows users to run event-driven and granularly billed applications, without having to address the operational logic" [10]. In this case, the user of the serverless platform might be the developer of a web application, and the operational logic could refer to bringing up the server, spinning up virtual machines, managing memory, and so on.

A few different services partially overlap with the definition above, such as Platform as a Service (PaaS), Function-as-a-Service (FaaS) and Software as a Service (SaaS). The primary factor that differentiates the three is the level of developer control. PaaS allows a developer to rent hardware resources and is billed by the hour. In this case, the developer has full control over the allocated resources and is responsible for managing them effectively. If servers sit idle due to a decrease in the workload, the developer will still be charged. On the other end of the spectrum is SaaS. SaaS is event-driven and granularly billed but does not run custom code. Instead, the developer can run prewritten service code. FaaS sits somewhere between PaaS and SaaS. The infrastructure is shared, but the application code is customizable. The developer is able to register modular pieces of custom code with the FaaS provider and set up triggers which will execute the code in response to an event, such as a user clicking a button in a web application. The term

serverless is most commonly used to refer to FaaS, which is the model I will focus on in this paper [6].

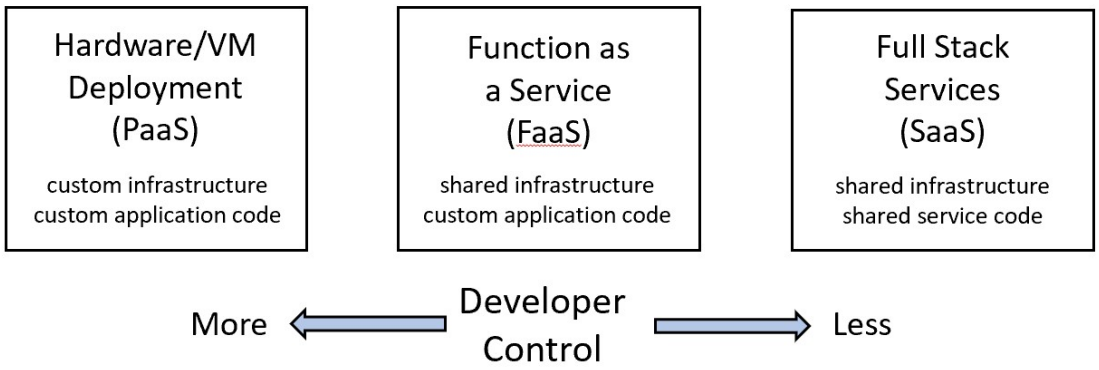

Figure 1: Level of developer control in serverless computing. Figure adapted from Baldini et al [6].

FaaS has a couple of key advantages over the traditional application development model. Resources are allocated by the service provider as needed, and developers are only charged for the execution time of their functions making it a highly scalable and cost-effective solution for enterprise application development [7]. This model is especially attractive due to the recent shift of application architectures to containers and microservices [6].

It should be noted that there are several important disadvantages to FaaS. For example, functions must be stateless and short. Most providers limit the execution time to about 10-15 minutes [4, 5, 18]. I will not explore these issues in this paper, but developers should be aware of all them when determining whether FaaS is suitable for their application.

Several providers currently support FaaS including AWS Lambda, Microsoft Azure, IBM Cloud Functions, and Google Cloud Functions. All meet the requirements of

FaaS by being event-driven and granularly billed, but may be implemented very differently [15]. It is important to understand how the internal configuration used by each FaaS provider might affect performance and cost. While Villamizar et al. have shown that there are clear financial benefits to using FaaS, relatively little is known about performance [26].

PRIOR WORK

Several studies have been performed, in an attempt to understand FaaS performance. The primary factors which have been shown to affect performance are cold start latency, programming language/runtime, and FaaS provider. There is also an overhead associated with running a function in a container rather than natively.

FaaS Cold Start

For security reasons, a new container must be started for each new function run on a particular machine. Containers may be reused, but only for exactly the same function registered by the same developer. Also, because resources are only allocated as needed, unused containers are shut down after a short grace period. This means that if functions are triggered infrequently, the containers will have to be unpaused or restarted every time the function is run.

Past work has shown that cold starts typically have a negative effect on latency, but the magnitude of the slowdown varies depending on provider and language. All providers and languages exhibit a cold start latency of at least 200ms, but in some cases the latency can be as long as 24s [13, 16, 24, 27]. This could be up to 10x the execution time of extremely short functions [24].

Effect of Programing Language and Runtime

Several studies have attempted to understand the effect of language on function performance. However, results vary widely depending on language and platform, and there is not yet enough information to come to a general consensus about the effect of language on performance for each platform.

Manner et al. tested functions written in Java and JavaScript on AWS Lambda and Microsoft Azure. They found that in both cases Java incurred a larger cold start overhead, but performed much better than JavaScript in a warm container [16]. They suggested that the larger cold start overhead was due to the virtual machine warmup time required by Java. The second result, Java outperforming JavaScript in a warm container, was expected since Java is a compiled language and JavaScript is interpreted. The primary difference between compiled and interpreted languages is that compiled languages do some amount of compilation before runtime, while interpreted languages compile or interpret high level code at runtime. Typically, compiled languages are faster since some of the work of translating high level code to machine readable instructions is done prior to runtime. However, based on previous FaaS research, this principle does not always hold.

Jackson et al. tested the effect of language runtime on AWS Lambda and Microsoft Azure. They found that on AWS Lambda Python outperformed all other languages, including Java, on warm starts [13]. Although Python is not strictly an interpreted language as it is compiled to bytecode which is then interpreted, it is typically considered interpreted because both steps occur at runtime. So, this result conflicts with the principle mentioned above.

It is also clear that different platforms are tuned for different languages. Jackson et al. found that C# .NET performed best on Microsoft Azure, but performed badly, particularly in cold start scenarios on AWS. NodeJS exhibited the exact opposite behavior,

performing well on AWS and poorly on Azure [13]. These differences are not surprising as providers are motivated to improve the performance of the most popular languages for their platform. However, it is extremely important for developers to understand how their choice of language and platform may impact function performance.

Containerization Overhead

Very little is known about the total slowdown caused by FaaS when compared to native execution. Sharad et al. found that for functions written in Python, there is a significant slowdown, especially if a cold start occurs. They observed up to a 12x slowdown for warm start scenarios and up to a 20x slowdown for cold start scenarios [24].

FaaSProfiler

Because not all FaaS platforms are opensource, most experiments to date have focused on reverse-engineering commercial FaaS systems. This can make it difficult to take reliable measurements as one cannot have control over the entire system. However, Sharad et al. recently performed a study on Apache OpenWhisk, the open-source FaaS platform used by IBM Cloud Functions. They deployed OpenWhisk on real servers, developed a profiling tool (FaaSProfiler) for invoking and collecting data on custom functions, and used that tool to study the architectural implications of FaaS [24].

MOTIVATION

Overall, prior work has shown very mixed results in terms of performance, and it is difficult to draw solid conclusions due to the black box approach taken by most studies so far. The FaaSProfiler designed by Sharad et al. helps to solve this problem by making benchmarking on Apache OpenWhisk simple and consistent.

The purpose of this report is to utilize this tool to further investigate the overhead due to containerization and cold starts for functions written in Java to determine whether prior findings regarding Apache OpenWhisk performance hold regardless of language.

In this study, I will measure the cold start latency for various Java functions. I will also determine the containerization overhead—relative to native execution—for both cold and warm starts. I expect to see a significant containerization overhead, especially for cold start scenarios. I would also like to determine whether Java functions incur an additional cold start overhead due to JVM warmup, on top of the container initialization time. Based on previous work, I expect to see a warmup overhead which is constant for the same function regardless of data size [14]. Finally, I will compare the containerization overhead experienced by Java functions to that of the same functions written in Python in order to understand the effect of language on OpenWhisk performance. Because previous OpenWhisk research has focused on Python and NodeJS, this comparison to Python can help place the results shown here among previous research.

In this chapter, I have given a brief introduction to my project. In chapter II, I will provide some background knowledge which is useful in understanding how FaaS works. Chapters III and IV contain the experimental methodology and results, respectively. Chapter V contains a discussion of my results and future work.

Chapter II: Background

In this section, I will give the reader a very basic background in writing and running functions on the OpenWhisk Architecture.

OpenWhisk Architecture

Just like all other FaaS architectures, OpenWhisk executes function code provided by a developer in an isolated environment—in this case, inside a docker container [9]. Because each invocation of the function is triggered by an event, the function code must be registered ahead of time so that when the event occurs, the code is already available in CouchDB—the database used for storing function code and results [2]. A docker container running NGINX receives and processes events which trigger an invocation request for a particular function [20]. That request is passed to the controller which locates the function in CouchDB. After the function code is retrieved, Kafka is used to queue requests and schedule the function invocation when enough resources are available [3]. Finally, a new docker container is started, if necessary, and the invoker runs the function. The result is returned to the database and the application which triggered the function.

Other FaaS providers have slightly different configurations, but all of them initiate function execution using triggers, execute functions in containers, and spin up or shut down those containers based on the invocation rate of functions and the total load on the server.

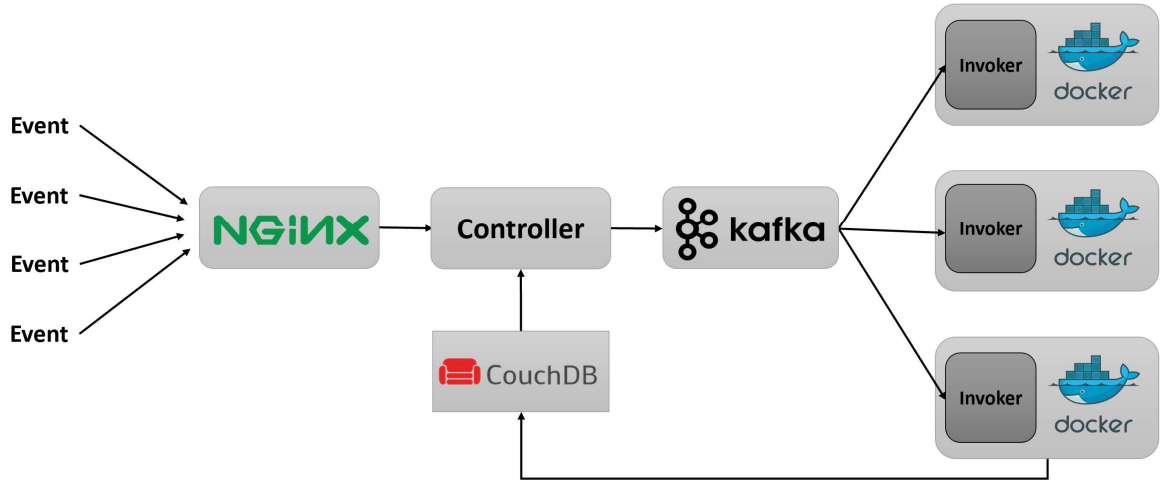

Figure 2: OpenWhisk architecture. Figure adapted from https://openwhisk.apache.org/documentation.html [4].

FUNCTION EXAMPLE

The benefits of FaaS computing have been described in detail above. Here I will describe how a function is actually written and invoked. It is important to note that functions must be modular and stateless—not dependent on any other piece of code. However, input parameters may be provided via command line inputs or a JSON file. On OpenWhisk, these modular pieces of code are called actions, so I will refer to them as such throughout the rest of this tutorial.

I will walk through a simple example of how an action is written, registered with OpenWhisk, and invoked. This example is adapted from the Apache OpenWhisk documentation, which can be found at the following url https://openwhisk.apache.org/documentation.html#python [4].

Below is a greeting action written in Python. It simply returns a JSON object with a greeting containing a name, if one is provided, or the default value “stranger” if no input is given.

```
def main(dict):
    if 'name' in dict:
        name = dict['name']
    else:
        name = "stranger"

    greeting = "Hello " + name + "!"

    return {"greeting": greeting}
```

The following line is used to register the function with OpenWhisk using the command line interface.

```
$ wsk action create helloPy hello.py -i
```

A confirmation message appears, indicating that the action has been successfully created. This means the code is in the database and the function can be invoked at any time.

```
ok: created action helloPy
```

The function can be invoked directly using the action invoke command. This is the manual alternative to setting up triggers.

```
$ wsk action invoke helloPy --blocking --param name Jane -r
-i
```

The greeting returned uses the name provided in the command above.

```
{
    "greeting": "Hello Jane!"
}
```

If no input parameter is provided, the default value is used.

```
$ wsk action invoke helloPy --blocking -r -i
```

The following output is produced.

```
{
    "greeting": "Hello stranger!"
}
```

Viewing all running docker containers, shows that a new container has been created for this action.

```
CONTAINER ID        IMAGE                                  COMMAND
                    CREATED                 NAMES
71A57A6F4DE7        openwhisk/python3action:nightly        "/bin/bash
-c `cd py…"         9 seconds ago           wsk00 6 guest helloPy
```

Both invocations of the greeting action shown above were run in this container. Invocations of the same function registered by the same user can share a container in order to save the time taken by a container start up. New containers are only started when: 1) a new function is run, 2) there are too many invocations of the same function for one container to handle, or 3) a function is run for the first time in several minutes and it's previous container was shut down.

Chapter III: Methodology

OpenWhisk and FaaSProfiler Configuration

A local deployment of OpenWhisk was created using public source code on GitHub and built from commit 81ac503 [1]. The only modification made from that commit was to increase the limits listed below from their default values. These limits can be found in the file located at "openwhisk/ansible/group_vars/all" and must be modified before deployment. Detailed instructions on deploying OpenWhisk can be found on the GitHub page. In these experiments, the instructions for native deployment on Ubuntu using Ansible and CouchDB were followed.

Parameter	Default Limit	Limit Used
invocationsPerMinute	60	60000
concurrentInvocations	30	30000
firesPerMinute	60	60000
sequenceMaxLength	50	50000

Table 1: Limits used in OpenWhisk deployment

After deployment, the OpenWhisk CLI was used to invoke OpenWhisk functions from the command line. Details on installing and configuring the CLI are included in the instructions mentioned above. The local configuration was used. After the CLI was set up, functions could be invoked using the wsk command as shown in Chapter II.

In order to automate OpenWhisk profiling, the FaaSProfiler was downloaded from GitHub, and commit c31d682 was used [21]. The only major modifications made were to config files used to set invocation rate, function parameter files, and so on. Parameters set in the config file are shown in Table 2. The values set for these parameters for each test are included in the appendix.

Parameter	Description
test_name	Name of tests to be run (can choose anything)
test_duration_in_seconds	Total duration of test in seconds, measurements stop after this time
random_seed	Ensures run to run consistency
blocking_cli	True/false determines whether blocking CLI calls are used
instances	Set of functions to run during measurement period
application	Name of function (should be exactly the name that is registered with OpenWhisk)
distribution	Distribution of function invocations
rate	Number of function invocations per second
activity_window	Range of time in seconds during which function invocations should occur
perf_monitoring	Set of scripts to be run during or after test
runtime_script	Monitoring script run during test
post_script	Optional post processing script

Table 2: Parameters in FaaSProfiler configuration files.

MICROBENCHMARKS

Four microbenchmarks were provided in the FaaSProfiler repository: primes, base64, http-endpoint, and json [21]. The functions provided were written in Python, JavaScript, Ruby, and Swift. I converted the four functions to Java and collected data for both the Python and Java versions for comparison. I also utilized five benchmarks from the Java Microbenchmark Harness repository [8]. Since these benchmarks were written for native execution, I converted them to OpenWhisk functions written in both Java and Python. Descriptions of each of the functions are given in Table 2. A data size is also defined for each function so that the reader can understand what was modified during the data scaling experiments which will be explained below.

Function Name	Description	Data Size (n)
primes	Finds the number of primes between 1 and n	Upper bound on range of numbers for prime search
base64	Encodes and decodes a string	Length of the string
http-endpoint	Performs API call to retrieve current time	Number of times API call is made
json	Reads a JSON object from a file and averages values with common fields	Length of the JSON object
big-decimal	Creates array of BigDecimals and compares all elements to element 0	Number of compare operations performed
big-integer	Creates array of BigIntegers and multiplies each element together	Number of elements in array
array-copy	Copies an array of bytes to an empty array (deep copy)	Length of byte array
file-write-read	Writes to tmp file and then reads data back	Number of bytes written to file
integer-max	Finds maximum value in an array of integers	Length of integer array

Table 3: Descriptions of functions used as microbenchmarks. Data size (n) is used in the data scaling experiments.

EXPERIMENTS

The goal of the experiments described below was to answer four primary questions:

1. What is the containerization overhead experienced by Java functions run on OpenWhisk?
2. What is the impact of a container cold start for functions written in Java and invoked on OpenWhisk?
3. Is execution time longer in a cold start scenario due to the JVM warming up?
4. How does Java performance on OpenWhisk compare with Python?

OpenWhisk vs. Native Performance

I measured the performance of each function run on OpenWhisk in both cold and warm start scenarios. In order to understand the containerization overhead, I ran the same functions natively using the Java Microbenchmark Harness (JMH) [8]. JMH ensures that the benchmarks are run in a warm environment and that no dead code is optimized out. I chose to collect the native execution data in a warm environment because this is more realistic to the traditional programming model, which constructs and executes one monolithic application. On the other hand, in FaaS a warm environment is never guaranteed as containers are spun up and shut down frequently. The difference in function execution times, including any container initialization, characterizes the total overhead caused by FaaS.

Cold Start

The cold start overhead was measured by running each of the functions mentioned above using the FaaSProfiler. The container initialization time and the function execution time were recorded. Each run was classified as cold or warm based the on the container initialization time. If the initialization time was greater than zero, it meant a new container had to be started and the run was considered cold.

Java Warmup and Data Scaling

In order to determine whether Java functions had longer execution times during cold starts, I measured the average execution time—not including container initialization—of each function for cold and warm runs. If the difference in the execution times was due to the JVM warming up, it should be consistent for the same function across data sizes. So, I also performed a data scaling experiment on each function by varying data size as described in Table 2.

Java vs. Python OpenWhisk Performance

To give context to the data collected in the previous few experiments, I measured OpenWhisk performance for the same functions—described in Table 3—written in Java and Python. I compared container initialization time as well as execution time in cold and warm start scenarios.

Chapter IV: Results

NATIVE VS. OPENWHISK EXECUTION

Function execution on OpenWhisk was consistently slower than native execution in both cold and warm start scenarios. Figure 3 shows the execution time of each function on OpenWhisk normalized by its native execution. Most functions experience an execution time between 2x and 5x slower than native execution during warm starts and between 5x and 10x slower during cold starts. However, extremely short functions such as json and big-integer experience much greater slowdowns, up to 50x. This is partially due to the constant overhead caused by container start up and initialization. Longer functions such as base64 and file-write-read are affected less as the container start represents a smaller percentage of their total latencies.

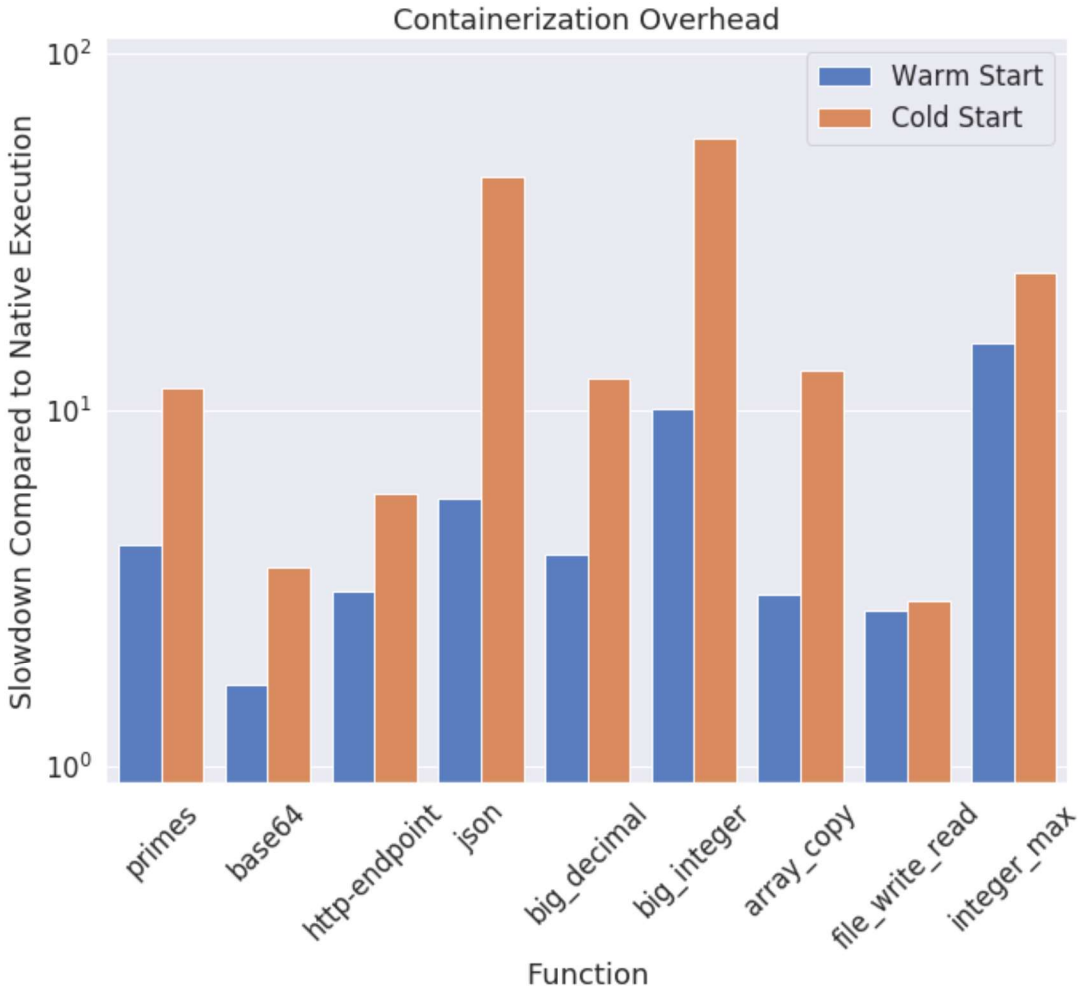

Figure 3: OpenWhisk execution time normalized by native execution time for Java functions.

COLD START

As shown in Figure 4, container initialization time was relatively constant across functions. This was expected because the time to start a container should not depend on the code it will run. Figure 5 indicates that container initialization time is not the only factor contributing to a functions' total latency in a cold scenario because the difference between

the latency on cold and warm starts is greater than the constant container initialization time, in most cases.

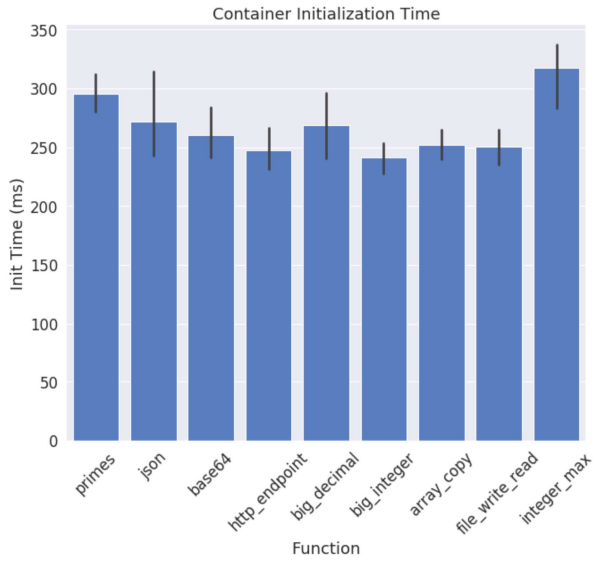

Figure 4: Container initialization time for Java functions.

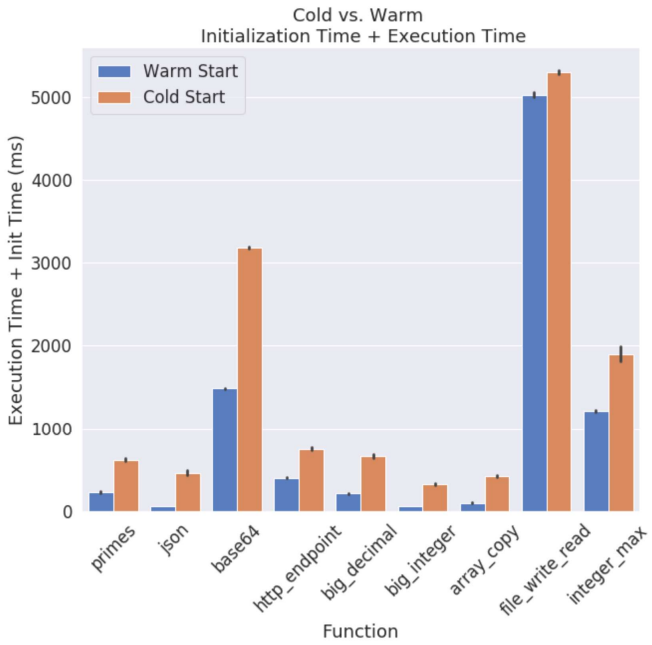

Figure 5: Sum of initialization time and execution time for Java functions.

I hypothesized that there would be an additional slowdown—on top of container initialization—due to JVM warmup. Figure 6 supports this hypothesis. It shows the execution time only, excluding initialization time, for functions written in Java as well as Python. For the Java functions, we can see that there is a clear difference in the execution time in cold and warm start scenarios, while Python functions always have the same execution time.

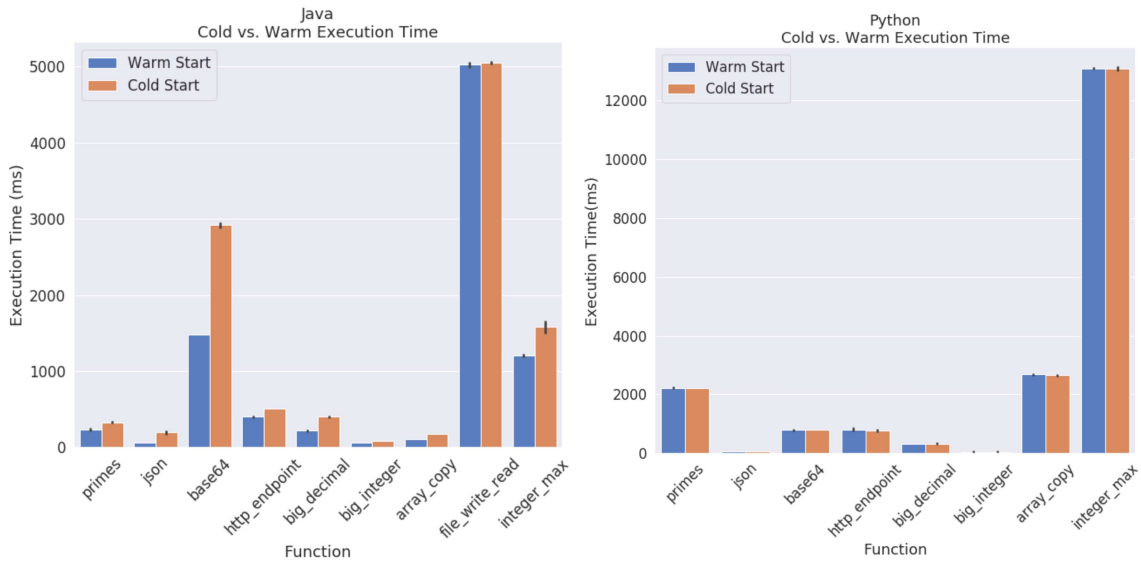

Figure 6: Execution time for Java and Python functions.

DATA SCALING

Assuming that the difference in the execution times between cold and warm runs is due to the JVM warming up, the overhead should be consistent for each function across various data sizes. Figure 7 confirms that the overhead is in fact consistent as long as the workload remains balanced. If the functions are over-invoked, meaning they are invoked too frequently causing the average function latency to continuously increase, the average runtime and the variation between runs will increase. I have not investigated the effects of

over- or under-invoked workloads because they have been explored in prior works [21].

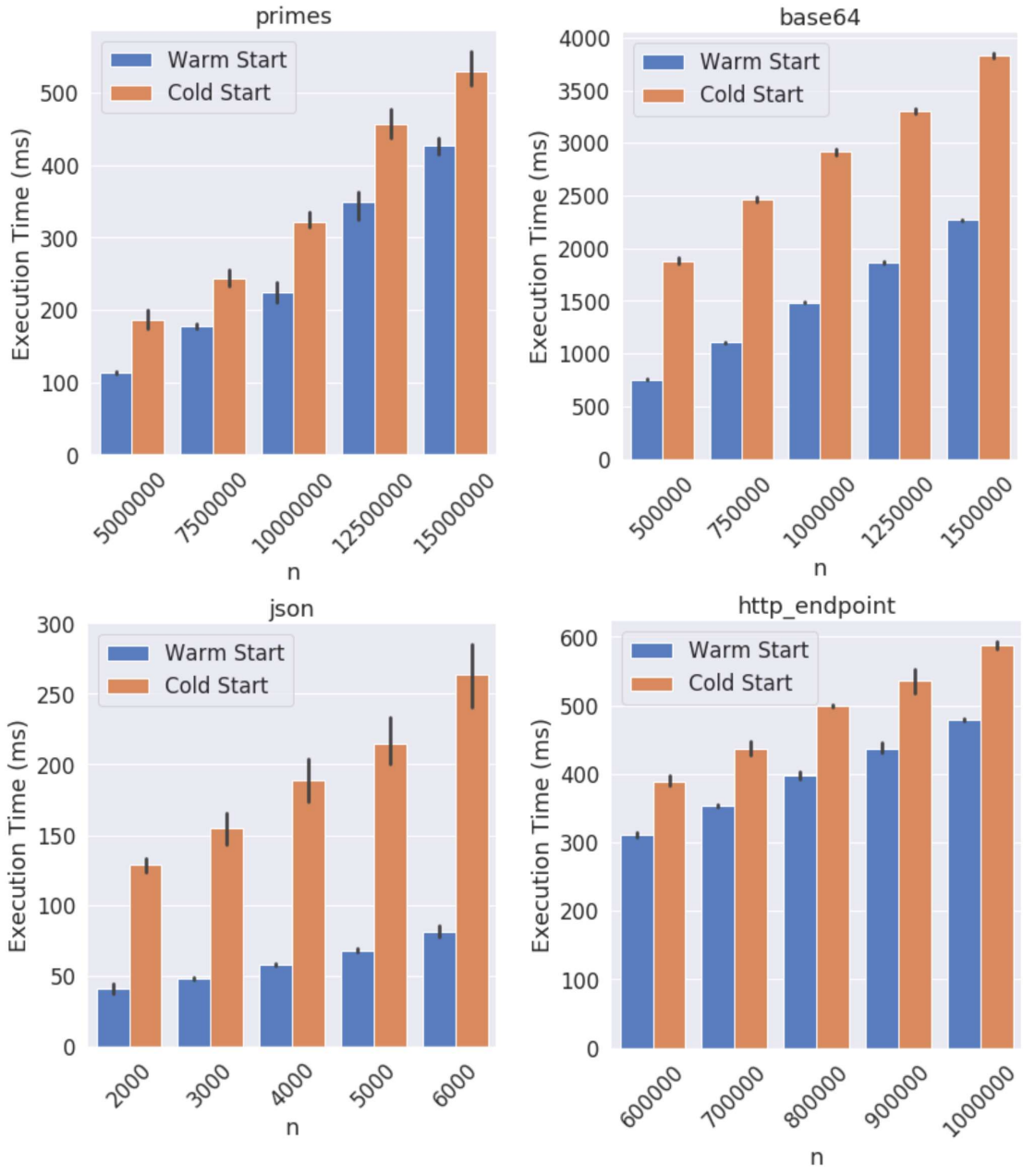

Figure 7: Data scaling experiments, where n is the data size as described in Table 2.

JAVA VS. PYTHON OPENWHISK PERFORMANCE

Figure 8 shows a comparison of Java vs. Python functions in cold (left) and warm (right) execution environments. Note that the y-axis is shown in log scale to accommodate functions with relatively large runtimes. Both plots show total latency, the sum of initialization time and execution time. Because Python has a longer initialization time than Java, the Java function is faster in all but one case for the cold start scenario. For the warm start scenario, Java is still faster on average, but there is more variation between functions. On average, Java is 3.6 times faster than Python in cold start scenarios and 6.7 times faster than Python in warm start scenarios. The larger gap between the two languages in the warm start scenario occurs because both the execution time and initialization time for Java functions are affected by cold starts. For Python, the initialization time is gone for warm starts, but the execution time remains the same.

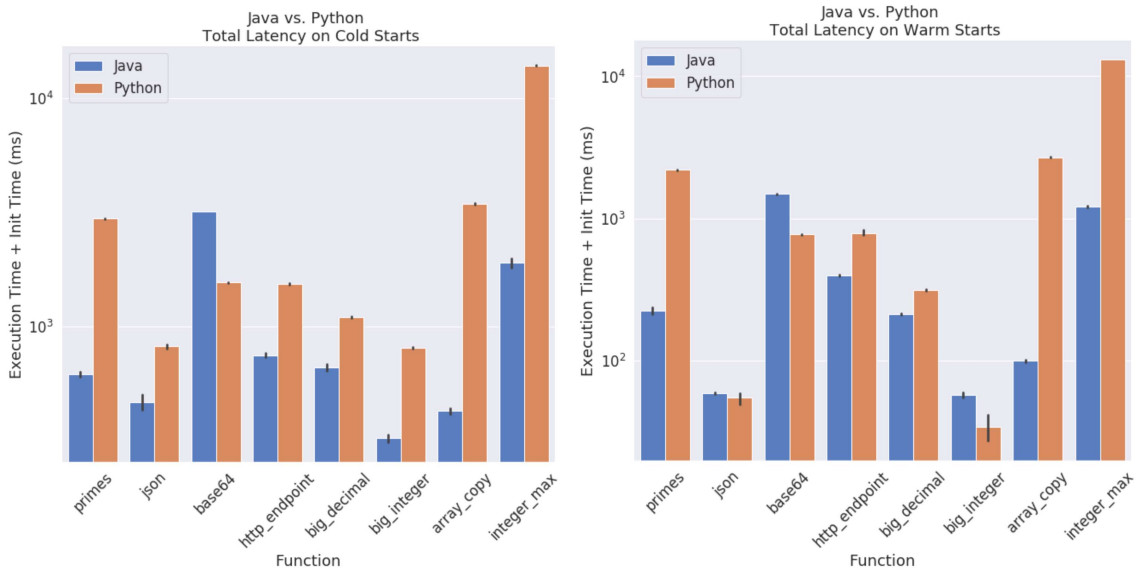

Figure 8: Total latency for Java vs. Python function on cold and warm starts.

Chapter V: Conclusion

TAKEAWAYS FOR THE DEVELOPER

This report provides three key takeaways for the FaaS developer. First, function runtimes should be significantly longer than the container initialization time. In general, short functions have a higher containerization overhead than long functions, and especially in cold start scenarios. Writing functions with longer runtimes—breaking up application code into larger pieces—can reduce the total containerization overhead. This can also help reduce the impact of cold starts because the constant container initialization time is incurred less frequently and represents a smaller percentage of the function's total latency.

Second, functions which are not invoked frequently enough will induce more cold starts. The cost of these cold starts varies depending on language. The results presented here show that the container initialization time for Python functions is higher than that of Java. However, Java functions typically incur an additional slowdown in the function's execution time on cold starts, while Python functions have consistent execution times across cold and warm starts.

Third, different FaaS platforms are tuned to different language runtimes. In other words, the language that performs well on one platform may perform very badly on another because the FaaS providers have incentive to tune their platform to the most popular language among their users. This is extremely important for developers to understand as choosing the appropriate planform/language pair can have a major impact on performance. In the case of OpenWhisk, Java performs better, even considering the additional overhead caused by JVM warmup during cold starts.

FUTURE WORK

There are several interesting areas of research in FaaS computing which could extend this work. I have shown that function length affects containerization overhead, and that typically longer functions perform better. However, additional research should be done to determine the optimal function length for any language or FaaS architecture. Similar studies on memory consumption and invocation rate should also be conducted.

There are some strategies already in place to attempt to improve cold start latency, but their exact impact on performance is unknown. Investigating the effect of pre-warming and other strategies used to reduce or eliminate container initialization time could help to quantify the performance improvement these methods provide.

Finally, most FaaS research to date has been focused on the performance of very short functions. Understanding the FaaS overhead for larger applications would be extremely helpful in determining the range of use cases where FaaS can provide a cost or performance benefit.

Appendix

This appendix includes a description of parameters used in FaaSProfiler configuration files. Below is an example of a configuration file, which specifies a single OpenWhisk function called “primesJava” with an invocation rate of 15. The total test duration is 15 seconds and the activity window is [5, 10]. This means data will be collected for a total of 15 seconds and the function will be invoked 15 times per second in the interval between the 5th and 10th seconds of the test.

```
{
    "test_name": "java_tests",
    "test_duration_in_seconds": 15,
    "random_seed": 100,
    "blocking_cli": false,
    "instances":{
        "instance1":{
            "application": "primesJava",
            "distribution": "Uniform",
            "rate": 15,
            "activity_window": [5, 10]
        }
    },
    "perf_monitoring":{
        "runtime_script": "monitoring/RuntimeMonitoring.sh",
        "post_script": null
    }
}
```

The only modifications made to config files between tests were to the test duration, application, rate, and activity window. These parameters were adjusted for each function in order to keep the system balanced. A balanced system is defined as one where the wait time—time the function spends waiting in a queue before it is allowed to run—does not

increase throughout the test. Previous work has shown that if a system is under- or over-invoked run-to-run variance increases drastically [21].

The following table shows the parameters used for each experiment described in Chapters III and IV.

Function	Parameter	Value (java tests)	Value (python tests)
primes	test_duration_in_seconds	15	15
	rate	15	5
	activity_window	[5, 10]	[5, 10]
base64	test_duration_in_seconds	30	15
	rate	5	8
	activity_window	[5, 25]	[5, 10]
http-endpoint	test_duration_in_seconds	15	15
	rate	15	15
	activity_window	[5, 10]	[5, 10]
json	test_duration_in_seconds	30	15
	rate	15	15
	activity_window	[5, 25]	[5, 10]
big-decimal	test_duration_in_seconds	30	30
	rate	15	15
	activity_window	[5, 25]	[5, 25]
big-integer	test_duration_in_seconds	15	15
	rate	15	15
	activity_window	[5, 10]	[5, 10]
array-copy	test_duration_in_seconds	20	60
	rate	10	2
	activity_window	[5, 15]	[5, 50]
file-write-read*	test_duration_in_seconds	60	60
	rate	1	1
	activity_window	[5, 50]	[5, 50]
integer-max	test_duration_in_seconds	90	90
	rate	5	1
	activity_window	[5, 85]	[5, 60]

* file_write_read was not measured for python because the parameters could not be adjusted to keep the machine in a balanced state and stabilize the execution time.

Table 4: FaaSProfiler configuration parameters by function.

References

[1] Apache, “apache/openwhisk,” *GitHub*, 17-Jan-2020. [Online]. Available: https://github.com/apache/openwhisk. [Accessed: 17-Jan-2020].

[2] *Apache CouchDB*. [Online]. Available: https://couchdb.apache.org/. [Accessed: 02-Apr-2020].

[3] *Apache Kafka*. [Online]. Available: https://kafka.apache.org/. [Accessed: 02-Apr-2020].

[4] *Apache OpenWhisk*. [Online]. Available: https://openwhisk.apache.org/. [Accessed: 31-Dec-2019].

[5] *AWS Lambda*. [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 31-Dec-2019].

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, “Serverless Computing: Current Trends and Open Problems,” *Research Advances in Cloud Computing*, pp. 1–20, 2017.

[7] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, “The serverless trilemma: function composition for serverless computing,” *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, 2017.

[8] “Code Tools: jmh,” *OpenJDK*. [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/. [Accessed: 17-Jan-2020].

[9] *Docker*. [Online]. Available: https://www.docker.com/. [Accessed: 02-Apr-2020].

[10] E. V. Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, “Serverless is More: From PaaS to Present Cloud Computing,” *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[11] L. Feng, P. Kudva, D. D. Silva, and J. Hu, "Exploring Serverless Computing for Neural Network Training," *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[12] *IBM Cloud Functions*. [Online]. Available: https://cloud.ibm.com/functions/. [Accessed: 30-Dec-2019].

[13] D. Jackson and G. Clynch, "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions," *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.

[14] D. Lion, A. Chiu, H. LiSun, X. Zhuang, N. Grcevski, and D. Yuan, "Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems," *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pp. 383–400, 2016.

[15] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.

[16] J. Manner, M. Endreb, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.

[17] G. Mcgrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017.

[18] *Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/. [Accessed: 30-Dec-2019].

[19] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in 11th USENIX Workshop on Hot

Topics in Cloud Computing (HotCloud 19), (Renton, WA), USENIX Association, July 2019.

[20] *NGINX*. [Online]. Available: https://www.nginx.com/. [Accessed: 02-Apr-2020].

[21] PrincetonUniversity, "PrincetonUniversity/faas-profiler," *GitHub*, 09-Jan-2020. [Online]. Available: https://github.com/PrincetonUniversity/faas-profiler. [Accessed: 17-Jan-2020].

[22] H. Puripunpinyo and M. Samadzadeh, "Effect of optimizing Java deployment artifacts on AWS Lambda," *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017.

[23] B. Ruan, H. Huang, S. Wu, and H. Jin, "A Performance Study of Containers in Cloud Environment," *Lecture Notes in Computer Science Advances in Services Computing*, pp. 343–356, 2016.

[24] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture - MICRO 52*, 2019.

[25] K.-T. A. Wang, R. Ho, and P. Wu, "Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment," *Proceedings of the Fourteenth EuroSys Conference 2019 CD-ROM on ZZZ - EuroSys 19*, Mar. 2019.

[26] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 233–247, 2017.

[27] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in 2018 USENIX Annual Technical Conference (ATC 18), (Boston, MA), pp. 133–146, USENIX Association, 2018.

[28] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, “Practical partial evaluation for high-performance dynamic language runtimes,” *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 662–676, 2017.